

Advance C++ Topics

We briefly consider class assignment, exceptions, templates and iterators.

19.1 Class Assignment

Suppose we have defined a class `Foo`. If we write:

```
Foo *bar1 = new Foo();
Foo *bar2 = bar1;
```

then the pointer `bar2` points to the same instance of `Foo` that `bar1` does. In particular, there is only one instance floating around.

If we write:

```
Foo bar1, bar2;
// changes to the two
bar2 = bar1;
```

then `bar2` is now a copy of `bar1`. Unless you specify otherwise, this is done by the default assignment operator, which is a *shallow copy*—only the declared variables are copied. For example, if `Foo` contains the header pointer to a linked list, the pointer will be copied, but both the header in `bar1` and the one in `bar2` will point to the same `Node` in memory. This is usually not what you want.

To create your own assignment operator, start with

```
Foo & operator= (const Foo &other) {
    // make copies of other's members and assign them to this
    return *this;
}
```

The `this` pointer always refers to the object on which the member function is being invoked. (It has to return the object, so that `A=B=C` works.) Now, one should deallocate the old stuff in the object using the `delete` command first—however, if the user writes `bar=bar`, assigning an object to itself, there is a potential problem.

Note that all languages have the shallow/deep copy issue.

19.2 Exceptions

An *exception* is an unexpected event that occurs when the program is running. In C++, an exception is treated as an object. There are exceptions already defined; it is also possible to create new ones.

An exception is explicitly thrown using a `throw` statement. A `throw` statement must specify an exception object to be thrown. All exceptions that are thrown must be eventually caught. A method might not handle an exception but instead propagate it for another method to handle.

A `try` clause is used to delimit a block of code in which a method call or operation might cause an exception. If an exception occurs within a `try` block, then C++ aborts the `try` block, executes the corresponding `catch` block and then continues with the statements that follow the `catch` block. If there is no exception, the `catch` block is ignored.

Good practice says that one should state which functions throw exceptions. This is achieved by having a `throw` clause that lists the exceptions that can be thrown by a method. Write the exception handlers for these exceptions in the program that uses the methods.

19.3 Templates

Thus far we have defined a special type for each collection. Templates let the user of a collection tell the compiler what kind of thing to store in a particular instance of a collection. If we want an set that stores strings, we say

```
set<string> S;
```

After that, the collection is used just the same as before.

The C++ code then needs templates. It is common to use a single letter for the parameter class. For example, the `Node` class might be written:

```
template <class T>
class Node
{
    public:
        Node(T initData, Node *initNext) {
            data = initData;
            next = initNext;
        }
        T data;
        Node *next;
}
```

and this is invoked with

```
template <class U>
class List
```

```
{
    Node<U> *head;
```

It is standard to break the class heading over two lines. To a large extent, you can think of `Node<>` as just a new class type.

Note that one can write code assuming that the parameter (`T` or `U`) implements various operations such as assignment, comparison, or stream insertion. These assumptions should be documented! Note that the template code is not compiled abstractly; rather it is compiled for each parameter choice separately, and the `cpp` file is included at the end of the header file.

19.4 Iterators

The idea of iterators is simply wonderful. They allow one to do the same operation on all the elements of a collection. Creating your own requires learning more C++ syntax. Using iterators is standardized. This is especially useful in avoiding working out how many elements there are: the basic idea is element access and element traversal.

In the STL (standard template) library, many structures have iterators. While the paradigm is the same for each, each iterator is a different type. A C++ iterator behaves like a pointer; it can be incremented and tested for completion by comparing with a companion iterator. The actual value is obtained by dereferencing.

Note that `begin()` returns the first element in the collection, while `end()` returns a value beyond the last element: it must not be dereferenced.

```
int addup ( vector<int> & A ) {
    int sum = 0;
    vector<int>::const_iterator start = A.begin();
    vector<int>::const_iterator stop = A.end();
    for( ; start!=stop; ++start )
        sum += *start;
    return sum;
}
```

You can leave out the `const_` part. In the above case, the `vector` template class also implements subscripting; so one could write:

```
int addup ( vector<int> & A ) {
    int sum = 0;
    for(int i=0; i<A.size(); i++ )
        sum += A[i];
    return sum;
}
```

Iterators also come up if testing whether a structure contains a particular element. Rather than having a built-in boolean `contains` function, one does:

```
template < class E >
bool contains( set<E> & B , E target ) {
    return B.find ( target ) != B.end();
}
```

We do not discuss how to create our own iterators.

19.5 Sample Code: SimpleList

```
// SimpleList.h - wdg - 2008
// header for primitive List
#ifndef SIMPLE_LIST_H
#define SIMPLE_LIST_H

template<class T>
class SimpleList
{
public:
    SimpleList(int cap);
    void add(T item);
    T getItem(int position) const;
    SimpleList<T> & operator=(const SimpleList<T> & rhs);

private:
    int count;
    int capacity;
    T *A;    // will store array
};

#include "SimpleList.cpp"

#endif



---


// SimpleList.cpp - wdg - 2008
// implementation code for a list of doubles
// uses partially filled array
```

```

#include <stdexcept>
using namespace std;

template <class T>
SimpleList<T>::SimpleList(int cap) : count(0), capacity(cap), A(new T[cap])
{
}

template <class T>
void SimpleList<T>::add(T item)
{
    if( count == capacity )
        throw invalid_argument("No room in list");
    A[count]=item;
    count++;
}

template <class T>
T SimpleList<T>::getItem(int position) const
{
    if( position<0 || position>=count)
        throw invalid_argument("Invalid position");
    return A[position];
}

template <class T>
SimpleList<T> & SimpleList<T>::operator=(const SimpleList<T> & rhs)
{
    if ( this!= &rhs ) { // uses addressof operator so that compare pointers
        capacity = rhs.capacity;
        count = rhs.count;
        delete [] A;
        A = new T[capacity];
        for(int i=0; i<count; i++)
            A[i] = rhs.A[i];
    }
    return *this;
}

```

```

// TestSimpleList.cpp - an inadequate test program - wdg 2009
#include <iostream>
using namespace std;
#include "SimpleList.h"

int main( )
{
    SimpleList<double> B(5); // calls constructor
    B.add(0);
    B.add(3.1415);
    B.add(1.414);
    cout << B.getItem(0) << " " << B.getItem(1) << " " << B.getItem(2) << endl;
    SimpleList<double> C(10);
    for(int i=0; i<10; i++)
        C.add(i);
    B=C;
    cout << B.getItem(0) << " " << B.getItem(1) << " " << B.getItem(2) << endl;
    cout << B.getItem(-99); // causes exception
    return 0;
}

```